Title: SYSTEM-LEVEL FAULT-TOLERANCE IN LARGE-SCALE
PARALLEL MACHINES WITH BUFFERED
COSCHEDULING

Author(s): Fabrizio Petrini, 154601, CCS-3
Marion K. Davis, 117267, CCS-3
Jose Carlos Sancho, 195715, CCS-3

Submitted to: IPDPS 2004

Los Alamos
NATIONAL LABORATORY

System-Level Fault-Tolerance in Large-Scale Parallel Machines with Buffered Coscheduling

Fabrizio Petrini, Kei Davis and José Carlos Sancho

Performance and Architecture Laboratory (PAL)
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory, NM 87545, USA
{fabrizio, kei, jcsancho}@lanl.gov

Abstract

As the number of processors for multi-teraflop systems grows to tens of thousands, with proposed petaflops systems likely to contain hundreds of thousands of processors, the assumption of fully reliable hardware has been abandoned. Although the mean time between failures for the individual components can be very high, the large total component count will inevitably lead to frequent failures. It is therefore of paramount importance to develop new software solutions to deal with the unavoidable reality of hardware faults. In this paper we will first describe the nature of the failures of current large-scale machines, and extrapolate these results to future machines. Based on this preliminary analysis we will present a new technology that we are currently developing, buffered coscheduling, which seeks to implement fault tolerance at the operating system level. Major design goals include dynamic reallocation of resources to allow continuing execution in the presence of hardware failures, very high scalability, high efficiency (low overhead), and transparency—requiring no changes to user applications. Preliminary results show that this is attainable with current hardware.

Keywords: Failure characterization, fault-tolerance, checkpointing, large-scale parallel computers, operating systems, communication protocols.

1 Introduction

The insatiable demand for ever more computational *capability*—computational power available to a single application—for scientific computing has driven the creation of ever larger parallel machines, these now achieving tens of teraflops (Tflops). To achieve such performance these machines comprise thousands of computing nodes[1] with large memories and storage capacities on the order of tens of terabytes (Tbytes). As an example, the ASCI Q machine [4] at Los Alamos National Laboratory (LANL) has 8192 processors and delivers 20 Tflops.

Several projects have been recently launched by the U.S. Department of Energy (DOE), in concert with computer manufacturers, to develop much larger machines to meet the requirements of larger and higher-fidelity simulations. For example, the BlueGene/L [2] supercomputer is a jointly funded research partnership between IBM and the Lawrence Livermore National Laboratory (LLNL) designed to deliver 360 Tflops peak performance. On the horizon are massively parallel machines intended to deliver petaflop (Pflop) or multi-Pflop performance, for example the machines of the DARPA HPCS projects [7].

It is clear that future increases in performance in excess of those resulting from improvements in single-processor performance will be achieved through corresponding increases in size, that is, component count. For example, the BlueGene/L supercomputer will have 65,536 nodes containing 608,256 DRAM memory modules. The large total component count of these massively parallel systems will make any assumption of complete reliability entirely unrealistic: though the mean time between failure (MTBF) for the individual components (e.g., processors, disks, memories, power supplies, and networks) and the physical connections between them may be very high, the large count of the components in the system will inevitably lead to frequent individual failures.

Unfortunately the current state of practice for fault tolerance in such systems is such that the failure of a single component usually causes a significant fraction of the system to fail, and any application using that part of the system to fail. There are two reasons for this: (1) the components

---

[1] See www.top500.org.

1

of the system are strongly coupled, for example, the failure of a fan is likely to lead to other catastrophic failures due to overheating; (2) application state is not stored redundantly, and loss of any state is catastrophic. With the MTBF for today's 10-20 Tflops machines on the order of 10-40 hours [15] and application running times on the order of days or weeks, multiple failures can be expected during a single execution of a single application. The implication of all of this is that the usability of proposed, much larger multi-Pflops systems is highly questionable.

Effective mechanisms for system-wide fault tolerance will become of increasingly critical importance for making the next generations of extreme-scale supercomputers useful and productive. Taking as a premise that massive hardware redundancy is economically unfeasible, and, we posit, unnecessary, and similarly that application software redundancy is unnecessarily wasteful of resources, the responsibility falls entirely to the system software.

The system software must be resilient to failures, providing dynamic reallocation of resources to allow continuing execution of applications in the presence of hardware failures. Other considerations include cost of development (i.e. not attempting to write an operating system from scratch), cost of maintenance, transparency to the application programmer (and so directly supporting legacy codes), very high scalability, and high efficiency—low overhead in terms of both resource utilization and degradation of system performance. Our contribution is the demonstration that these goals are attainable with current hardware and without requiring user intervention. We anticipate that this will be feasible to implement at the system level with a very low overhead. Two recent publications provide the basis for this claim. One [19] demonstrates the feasibility of implementing incremental checkpointing and rollback mechanisms within the limitations imposed by current hardware. The other [10] showcases the new proposed communication technique, Buffered CoScheduling (BCS), that simplifies the production-level MPI implementations.

The rest of this paper is organized as follows. In Section 2 we describe the nature of the failures of current large-scale machines. Section 3 summarizes the current approaches to ameliorating or avoiding failures in clusters. Section 4 describes our proposed solution to provide fault-tolerant capabilities to large-scale machines. Section 5 provides some concluding remarks.

2 Failure Characterization

Improvements in design, manufacturing techniques and quality control, and testing of computer components have made for a historical trend of increased reliability concomitantly with increased performance and capacity. Components of commodity computers, such as processors, DRAM

modules, large capacity disks, power supplies, and fans have an operational lifetime measured in years. Perhaps the extreme example of increased reliability (and capacity) are today's processors: the MTBF for a high-quality unit is on the order of 1,000,000 hours (110 years).

Regardless of nominal MTBF, typically there is a phase in component lifetime where failure rates can be higher. Generally, the failure rate model for components follows a *bathtub curve* as shown in Figure 1. As can be shown, there are three different phases: component burn in, normal aging, and late failure. Failures are frequent during the burn in and the late failure phases due to defects in the components, and component aging, respectively.
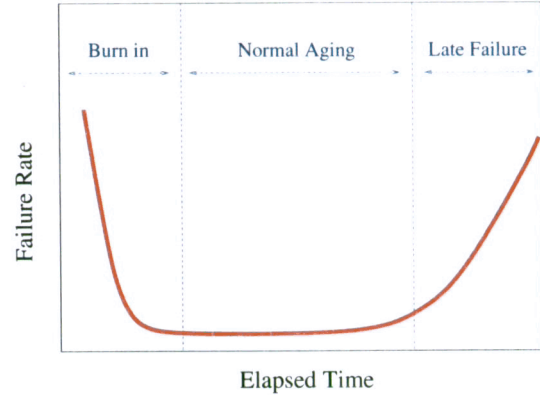

Figure 1. Component failure model.

For the sake discussion the MTBF of individual components is defined as: $MTBF = 1/\lambda$, where the parameter λ is called the failure rate of the component.[2] In a system where the failure of a single component results in the failure of the entire system the MTBF of the system is given by:

$$MTBF = \frac{1}{\sum_{i=1}^{i=N} \lambda_i}$$

Figure 2 shows the expected system MTBF obtained from this model for system for three different component reliability levels: MTBFs of 10^4, 10^5, and 10^6 hours. As can be seen, when increasing the number of components in the system the MTBF of the entire system dramatically decreases. For projected Pflop systems the MTBF is only a few hours even when *all* components are of very high reliability (MTBF 10^6 hours). In particular, for a system with 100,000 nodes the MTBF would be 10 hours. The BlueGene/L system with 65,536 nodes is expected to have an MTBF of less than 24 hours. In reality these numbers are optimistic for several reasons: MTBF numbers for individual components assume ideal environmental conditions, components will not generally have uniformly high MTBFs,

---

[2]This assumes that the lifetime of components is exponentially distributed.

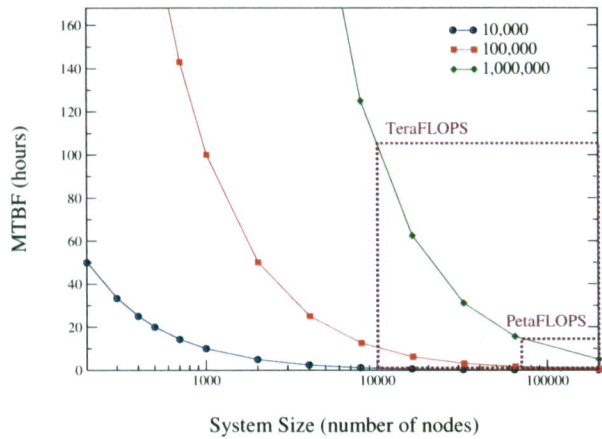and failure of some components can damage others, reducing their MTBFs.

for fault tolerance yet more important.

Figure 2. System MTBF as a function of system size and component MTBF.

Figure 3. Failure distribution for the ASCI Blue Mountain during 2003.

Factors contributing to failures in a computer system can be roughly divided into three main categories (1) *hardware failures*, on processing nodes or in the network, for example, failures in the CPU, memory modules, PCI bus, and interconnection network; (2) *software errors*, comprising failures in the cluster file system, compilers, libraries, operating system, MPI, and user code; and (3) *other failures*, such as human (operator) errors, environmental (cooling, power), soft errors, and miscellaneous undetermined errors.

Studies of the distribution of failures in current systems, such as the ASCI Blue Mountain [3] (3072 processors) and CPLANT [6], show that hardware failures predominate. Figure 3 shows the distribution of the factors contributing to system downtime for the ASCI Blue Mountain. Hardware failures account for 80.5 percent of the incidents resulting in downtime, whereas software failures and other causes represent only 4.9 and 15.1 percent, respectively. The hardware components failing most frequently are the memory modules, accounting for 37.8 percent, and the node boards and power supplies, at 14.5 and 11.7 percent, respectively. A similar distribution of failures has been reported for the CPLANT [8]. Figures for ASCI Q are qualitatively similar.

The profile of failure modes for supercomputers is consistent and distinct from other large computing systems. For example, in Internet services infrastructure, operator and software errors are the major contributor to system downtime [16]. We claim that not only will hardware failures be the major cause of failure in the next generation of supercomputers, but their contribution (as a percentage of all failures) will likely be greater, making effective mechanisms
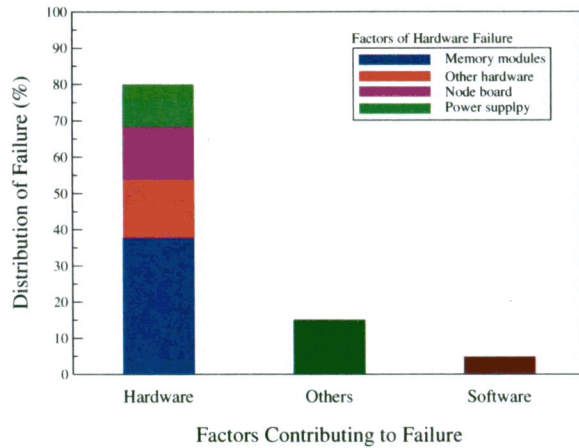
3 Dealing with Faults

Fault tolerance has become a central issue in the procurement of large-scale parallel machines. In fact, it is perceived that machines with tens of thousands of processors will not be able to effectively devote all of their processing power to large user applications.

Two commonly used measures of the overall productivity of large-scale parallel platforms are *capability* and *capacity*. The first measure, capability, refers to the maximum processing power that can be brought to bear on any one job. The second, capacity, represents the total processing power (possibly by more than one machine) available for running (multiple) applications. A given amount of capability implies at least as much capacity, but the converse is not generally true: to achieve full capacity may require running multiple jobs simultaneously. Today the ASCI platforms of highest capability are LLNL's White at 12.3 Tflops and LANL's Q at 20 Tflops. The next planned acquisitions are Sandia National Laboratory's Red Storm, projected to deliver 40 Tflops, and LLNL's Purple C at 100Tflops.

A recent report from the JASONs, a committee of distinguished scientists chartered by the Department of Defense (DoD) to advise the agencies of the US government on scientific issues, raises the sensitive question of whether future ASCI machines can be used as capability engines. For that to be possible major advances in fault tolerance are needed. Scaling to Pflop performance using present machine architectures implies that a very large number of processors, on

the order of 100,000, might be needed. Such large numbers raise questions of both scalability of application performance and of machine reliability. So, while there is a perceived need for Pflop capability within a decade, it appears that there is no clear path to Pflop software and hardware architecture. To the forward looking it is painfully obvious that research in these areas, or lack thereof, will predicate success or failure.

Notwithstanding the complexity of these problems, the ASCI program has managed to achieve a degree of fault tolerance through a set post-hoc measures.

User-Initiated Checkpointing. The typical approach to guaranteeing an acceptable level of fault tolerance is for the programmer to save the relevant state (data) of the application at regular intervals. This approach is rather simple, it only requires a parallel file system, but has several disadvantages. First of all, it requires effort and care by the programmer who must know the structure and properties of the parallel application and so opportunistically checkpoint the state during the execution of the program. This information is specific to each program and the transformation cannot be applied automatically.

Full checkpoints are in general very inefficient. They block the entire application for the potentially long time that is needed to save to nonvolatile storage the important part of the memory image of each process, which can amount to several GB. Also, full checkpoints stress the I/O subsystem in a bursty way, because the information is sent synchronously to storage. It is worth noting that the I/O generated by checkpointing, called *defensive I/O*, accounts for almost 80% of all I/O on ASCI class machines, and therefore influences the design of the machine by elevating the relative importance (size, speed, cost) of the I/O subsystem.

Finally, user-initiated checkpoints tend to exhibit coarse granularity, occurring every few hours. Thus, in the presence of a fault, the rollback can loose hours of useful work. With larger machines and decreased MTBF, to be effective checkpointing would need to occur more frequently; it is not hard to see that this could degenerate to an application spending all of its time just checkpointing.

Segmentation of the Machine. A procrustean approach to bounding the impact of faults is to segment a large-scale machine into independent subsets that are used by distinct applications. Obviously capability is correspondingly divided.

Removal of fault-prone components. Another approach is to eliminate components that are not absolutely necessary. Cluster management software utilizing a control network eliminates the need for floppy or optical drives on every node. More recent has been the elimination of hard

disks [12, 11]. However, lack of local non-volatile storage makes checkpointing yet more expensive. Also, one of the main culprits, DRAM, cannot be eliminated, and the absence of disks means that either ordinary caching to disk becomes more expensive or more DRAM must be present.

4 Proposed Solution

Our proposed solution is based on an automatic, frequent, user-transparent, and coordinated checkpoint and rollback-recovery mechanism. In essence, compute nodes are coordinated globally in order to checkpoint the computation state of the parallel program. In case of failure the non-functional portion of the machine is identified, a reallocation of resources is made, the compute nodes roll back to the most recently saved state, and the computation is continued.

A new and key aspect of this approach is that by enforcing *global coordination* the potentially undesirable *domino effect* [9], wherein the consistent recovery state may be very far from the current computation state, may be avoided.

Automatic and *user-transparent* checkpointing mechanisms are increasing in importance in large-scale parallel computing because the problems with user defined and managed checkpointing techniques only become worse with increasing system size. These problems are strongly interrelated but may be roughly divided into the categories of complexity, granularity, resource cost, programmer cost, and correctness.

Complexity. The complexity of the checkpoint state of a system consisting of thousands of processors is huge because of the large count of the computational states distributed between all the processors in the system; the complexity of the communication state can be exponential in the system size when processors are communicating asynchronously with one another.

Granularity. User-defined mechanisms are by practical necessity opportunistic—checkpointing is performed at program points where the state is most easily codified by the programmer; for adaptive codes, for example (e.g. AMR), these may be few and far between on a time scale, resulting in very coarse granularity.

Resource cost. User-defined mechanisms typically record states in their entirety at every step; user-defined incremental checkpointing is usually impractical.

Correctness. An automatic checkpointing mechanism, if correct, will be correct for all applications. User-defined mechanisms are error-prone, difficult to debug, and must be

crafted for every application and potentially tuned for every architecture.

• We claim that highly scalable, highly efficient, transparent, and correct fault tolerance may be obtained using two mechanisms:

- **Buffered CoScheduling.** that enforces a sufficient degree of determinism on communication to make the design of efficient, scalable checkpointing algorithms tractable; and,

- **Incremental checkpointing.** that exploits the determinism imposed by Buffered CoScheduling.

4.1 Buffered CoScheduling

Buffered CoScheduling MPI (BCS-MPI) [10] is a new approach to the design of the communication layer for large scale parallel computers. The innovative contribution of BCS-MPI is that it imposes a specific degree of determinism on the scheduling of the communication in parallel applications.

BCS-MPI orchestrates all communication activities at fixed intervals (timeslices) of a few hundreds of microseconds. At each interval communication is strictly scheduled: only the messages that can be delivered in a given interval and have been globally scheduled are injected to the network. Messages that require more than one interval are chunked in segments and scheduled over multiple intervals. The important aspect of this approach is that at the end of each interval the network is empty of messages. This guarantees that at certain known times during program execution there are no messages in transit; it is in this sense that determinism is imposed.

From the point of view of a checkpointing and rollback recovery mechanism this vastly simplifies the network state: the network is empty, all pending (portions of) messages are known, the remaining state is the set of memory images of the processes of the application, and the checkpointed data itself.

Figure 4 outlines the steps taken during a timeslice to perform scheduling of global communication.

There are two main phases:

1. **Global message scheduling phase.** The global communication scheduling for the next timeslice based on all the communication requests (message descriptors) generated by each application process and posted to the network interface card (NIC) during the previous timeslice. The scheduling is performed in two *microphases*: a descriptor-exchange microphase wherein a partial exchange of information between remote NICs is performed, and a message scheduling microphase wherein point-to-point and collective opera-
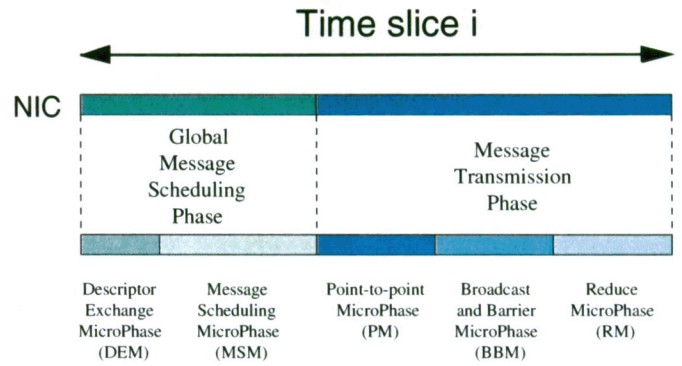

Figure 4. Global synchronization protocol of the BCS-MPI mechanism.

tions are scheduled using information obtained in the previous microphase.

2. **Message transmission phase.** The point-to-point and collective operation scheduled in the previous phase are actually performed. At the end of this phase all transmissions have been completed; no messages are in transit in the network.

The implementation of this mechanism is based on a set of communication primitives (BCS core) which are tightly coupled with the primitives provided at hardware level by the network. For example, the Quadrics network provides these primitives at the hardware level [17]. The hardware characteristics are such that these primitives, and BCS core, have very high scalability.

Figure 5 shows the library hierarchy of BCS-MPI. The BCS-MPI runtime system is implemented in terms of the BCS API, which in turn uses the BCS core primitives. The BCS API simply maps MPI calls to BCS calls. This approach enormously simplifies the design and maintainability of BCS-MPI.

The performance and feasibility of this mechanism has been evaluated and validated on a preliminary prototype implemented at user (i.e. not OS) level with most of the code running on the NICs. It is expected that implementation at the system level, e.g. by a Linux kernel module, will be faster.

Despite the constrained communication of BCS-MPI, preliminary results reported for synthetic benchmarks shows that the loss of performance of the application is less than 7.5 percent with a computation granularity of 10ms on 62 processors. Moreover, the slowdown significantly decreases when the computation granularity is increased because the delay introduced in communication is more than made up by increased time for computation.

Evaluation of the scalability of this mechanism has shown that the slowdown is almost unchanged by the num-
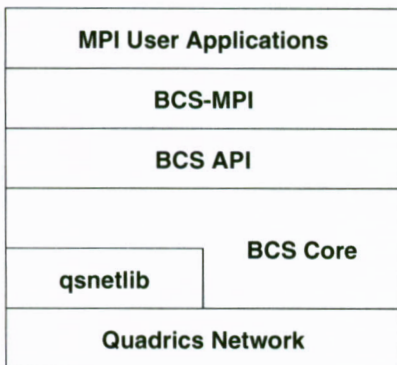
5

Figure 5. Library hierarchy of the BCS-MPI mechanism.

ber of processors in the system. The importance of this result is that it provides convincing evidence that BCS-MPI will exhibit very high scalability.

Evaluation of relative performance for real scientific applications shows that for some applications, such as Sage [14] and Sweep3D [21], the performance penalty is negligible, though for some NAS parallel benchmarks [5] the performance penalty may be as high as 15 percent. Nonetheless these results are very encouraging because Sage and Sweep3D are representative of a large class of ASCI production codes at LANL.

4.2 Incremental Checkpointing

The primary potential problem of frequent, automatic, and user-transparent checkpointing and rollback recovery is the quantity of generated checkpoint data. Frequent checkpointing of the large memory footprints of scientific applications can quickly saturate available bandwidth and fill nonvolatile storage.

Incremental checkpointing [18] is a well-known optimization to reduce checkpoint data. The optimization is achieved by only saving that part of the memory footprint that has changed since the previous checkpoint. Recent work [19] has shown that frequent, automatic, and user-transparent incremental checkpointing and rollback recovery mechanisms are achievable within the limitations of current hardware. That work evaluates the bandwidth, the *incremental bandwidth* (IB), required to save the changes to the memory, at the granularity of the operating system page, during the course of program execution. A major contribution of this work is the demonstration that incremental checkpointing may be realized without special hardware and without changes to application source code.

Preliminary experimental results have shown that the IB is sensitive to the checkpoint interval. The encouraging result is that larger checkpoint intervals substantially decrease

the IB requirement. To illustrate this Figure 6 shows the average and maximum IB reported for Sage using a memory footprint size of 1000MB with checkpoint intervals between 1s and 20s. The explanation for these results is that scientific applications tend to frequently reuse a small subset of pages of the memory footprint, hence longer checkpoint intervals result in a reduction in IB. Even for short checkpointing intervals the average IB reported is 78.8MB/s, which is a small fraction of the bandwidth of the interconnection network (900MB/s) [1], PCI-X busses (1GB/s) and hard disks (320MB/s) [20].
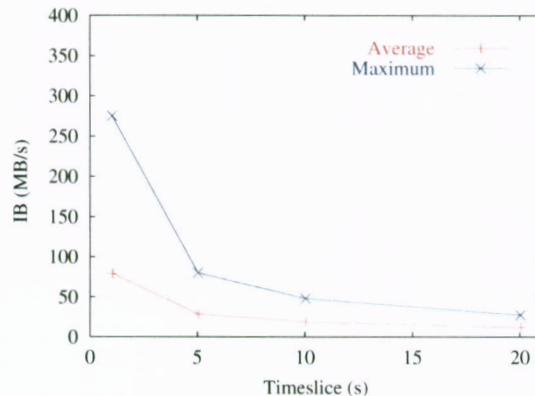

Figure 6. Maximum and minimum IB required for checkpointing Sage-1000MB.

Table 1. Bandwidth Requirements

Application	Maximum (MB/s)	Average (MB/s)
Sage-1000MB	274.9	78.8
Sage-500MB	186.9	49.9
Sage-100MB	42.6	15
Sage-50MB	24.9	9.6
Sweep3D	79.1	49.5
SP	32.6	32.6
LU	12.5	12.5
BT	72.7	68.6
FT	101	92.1

Similar results were achieved for other applications and memory footprint sizes, even with a relatively frequent checkpoint interval of 1s, as shown in Table 1 [19]. Moreover, that work shows that incremental checkpointing is scalable in terms of bandwidth requirements. Experiments performed for weak scaling, wherein the problem size grows proportionally with the number of processors, shows that the bandwidth requirements are almost invariant with

6

respect to processor count; for large processor count the bandwidth requirements are slightly lower.

Another important issue affecting scalability is the effect of increasing the memory footprint of a given application. Experimental results showed that the IB increases sublinearly with the memory footprint size. This result is illustrated in Table 1 for Sage. For example, the average IB obtained for Sage-500MB is 49.9 MB/s while that for Sage-1000MB is only 78.8MB/s. Note that employing a full checkpointing technique would result in bandwidth requirements increasing linearly with memory footprint size, so that not only does incremental checkpointing require less bandwidth and storage than full checkpointing, it is also more scalable.

Finally, it is worth pointing out that checkpointing techniques will become more efficient in the future because the speeds of networks, I/O busses, and storage devices are increasing at greater rates than main memory [13].

5 Conclusions

In this paper we outlined a new software solution for fault tolerance that will allow the next generation of extreme-scale parallel computers to be used as full-capacity capability machines. Our solution is based on recent work that provides strong evidence that a frequent, automatic, transparent, and incremental checkpointing and rollback recovery technique is realizable with current hardware.

First, experimental results indicated that the implementation of frequent, automatic, user-transparent incremental checkpointing is a viable technique within the current technology without requiring the support of specialized hardware and without changing the structure of the application source code. They also provide enough robustness to generalize the results to future large scale parallel computers because of the technological trends of networks and storage devices and the scalability properties of the incremental checkpoint technique.

Finally, BCS provides a solid infrastructure on which to implement scalable checkpointing mechanisms. BCS greatly simplifies the complexity of the communication state of parallel applications providing a scheduled, deterministic communication behavior. Indeed, the prototype implementation of this system software support proofs that those advances can be achieved with a negligible performance penalty.

The fault tolerant software solution consisting of incremental checkpointing and BCS can be implemented at system level with minimal intrusiveness (low overhead) and complete transparency.

References

[1] D. Addison, J. Beecroft, D. Hewson, M. McLaren, and F. Petrini. Quadrics QsNet II: A network for Supercomputing Applications. In *Proceedings of the Hot Chips 14*, Stanford University, California, August 18–20, 2003. Available from http://www.c3.lanl.gov/~fabrizio/talks/hot03.ppt.

[2] N. R. Adiga and et al. An Overview of the BlueGene/L Supercomputer. In *Proceedings of the Supercomputing 2002, also IBM research report RC22570 (W0209-033)*, Baltimore, Maryland, November 16–22, 2002. Available from http://sc-2002.org/paperpdfs/pap.pap207.pdf.

[3] ASCI Blue Mountain. Available from http://www.lanl.gov/asci/bluemtn.

[4] ASCI Q machine. Available from http://www.lanl.gov/asci/.

[5] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. NAS 95-020, NASA Ames Research Center, Moffett Field, California, December 1995. Available from http://www.nas.nasa.gov/Research/Reports/Techreports/1995/nas-95-020-abstract.html.

[6] CPLANT. Available from http://www.cs.sandia.gov/cplant/.

[7] High Productivity Computing Systems (HPCS) initiative in DARPA. . Available from http://http://www.darpa.mil/ipto/programs/hpcs/index.html.

[8] D. Doerfler. Architectural considerations in delivering a balanced linux cluster. In *Proceedings from the Conference on High Speed Computing*, Gleneden Beach, Oregon, April 22–25, 2002. Available from http://www.ccs.lanl.gov/salishan02/doerfler.pdf.

[9] E. N. Elnozahy, L. Alvisi, D. B. Johnson, and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, September 2002. Avail-

able from `ftp://ftp.cs.cmu.edu/user/mootaz/papers/S.ps`.

[10] J. Fernández, E. Frachtenberg, and F. Petrini. BCS MPI: A New Approach in the System Software Design for Large-Scale Parallel Computers. In *Proceedings of SC2003*, Phoenix, Arizona, November 10–16, 2003. Available from `http://www.c3.lanl.gov/~fabrizio/papers/sc03_bcs.pdf`.

[11] Manish Gupta. Challenges in developing scalable scalable software for bluegene/l. In *Scaling to New Heights Workshop*, Pittsburgh, PA, May 2002.

[12] Erik Hendriks. BProc: The Beowulf distributed process space. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing (ICS '02)*, New York, NY, June 22–26, 2002.

[13] J. L. Hennessy, D. A. Patterson, and D. Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2002.

[14] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Proceedings of the IEEE/ACM Supercomputing*, Denver, Colorado, November 10–16, 2001. Available from `http://www.sc2001.org/papers/pap.pap255.pdf`.

[15] D. J. Kerbyson, A. Hoisie, and H. J. Wasserman. Use of Predictive Performance Modeling During Large-Scale System Installation. In *Proceedings of the First Internnational Workshop on Hardware/Software Support for Parallel and Distributed Scientific and Engineering Computing*, Charlottesville, Virginia, September 22–25, 2002. Available from `http://www.c3.lanl.gov/par_arch/pubs/KerbysonSPDEC.pdf`.

[16] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet Services Fail, and What can be done about it? In *Proceedings of Usenix Symposium on Internet Technoloies and Systems*, Seattle, Washington, March 26–28, 2003. Available from `http://roc.cs.berkeley.edu/papers/usits03.pdf`.

[17] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002. Available from `http://www.c3.lanl.gov/~fabrizio/papers/ieeemicro.pdf`.

[18] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the Usenix Winter 1995 Technical Conference*, New Orleans, Louisiana, January 16–20, 1995.

Available from `http://www.cs.utk.edu/~plank/plank/papers/USENIX-95W.html`.

[19] J. C. Sancho, F. Petrini, G. Johnson, J. Fernández, and E. Frachtenberg. On the Feasibility of Incremental Checkpointing for Scientific Computing. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium*, Santa Fe, New Mexico, April 26–30, 2004. Available from `http://www.c3.lanl.gov/~fabrizio/papers/ipdps04.pdf`.

[20] SCSI hard drive Seagate model Cheetah. Available from `http://www.seagate.com/docs/pdf/datasheet/disc/ds_cheetah15k.3.pdf`.

[21] The ASCI Sweep3D Benchmark Code. Available from `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/`.